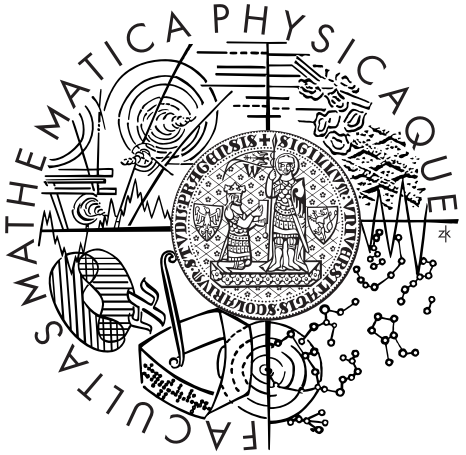
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Pavel Mach

Face Recognition in Social Networks

Department of Software Engineering

Supervisor of the bachelor thesis: doc. RNDr. Tomáš Skopal, Ph.D.

Study programme: Informatics

Specialization: Programming

Prague 2012

I would like to thank my supervisor, doc. RNDr. Tomáš Skopal, Ph.D., for help and valuable advices during my work. I would like to thank my parents for their patience during my studies. Last but not least, I thank my friends and relatives for letting me use their photos in the thesis.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Rozpoznávání tváří v sociálních sítích

Autor: Pavel Mach

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. RNDr. Tomáš Skopal, Ph.D.

Abstrakt: Rozpoznávání tváří je čím dál tím populárnější služba v různých obrázkových galeriích a sociálních sítích. Žádná z těchto reálných aplikací však neposkytuje uživateli možnost nastavení parametrů. Cílem této práce je vyvinout knihovnu pro detekci a rozpoznávání tváří kterou bude možno jednoduše použít v jiném programu a také webovou aplikaci rozpoznávající tváře, prostředím podobnou sociální síti, která bude tuto knihovnu využívat. Samozřejmě se nesnažíme konkurovat rozponávání tváří ve velkých sociálních sítích, mimo jiné proto, že to vzhledem k jejich obrovské databázi není dobře možné. V experimentální části práce se snažíme nalézt co nejvhodnější parametry, aby detekce i rozpoznávání byly co nejpřesnější.

Klíčová slova: rozpoznávání tváří, sociální sítě, eigenface

Title: Face Recognition in Social Networks

Author: Pavel Mach

Department: Department of Software Engineering

Supervisor: doc. RNDr. Tomáš Skopal, Ph.D.

Abstract: Popularity of face recognition in image galleries and social networks is growing. But none of these real world applications let the user adjust parameters. The goal of this thesis is to develop a library for face detection and recognition which can be easily used in some other program and also a web application recognizing faces, with environment similar to the one of social networks, which uses the library. Of course we do not try to compete with big social networks in the field of face recognition, because it will be almost impossible due to their enormous databases. In the experimental part of the work we try to find most suitable parameters such that face detection and face recognition would be as accurate as possible.

Keywords: face recognition, social networks, eigenface

Contents

Introduction

We implemented a library for face detection and recognition together with a sample application showing its capabilities. Also, we created a web application with an environment similar to social networks, where the library is used for face recognition. This work describes face recognition in photos as well as its implementation.

Computer vision

The growth of multimedia databases, together with increasing computing performance, has pushed demands on machines not just to interpret images or video, but actually to "understand" what they are representing. The application of computer vision is huge.

Modern cars are helping us by detecting pedestrians, recognizing road signs and Google is even working on a project of a driverless car. During tests, these cars drove 1,609 kilometers without any human intervention as an addition to 225,308 kilometers with occasional human interaction [11]. Another example of state-of-the-art usage of computer vision is NASA's Mars Exploration Rover.

Computer vision is applicable not only in these high-tech devices but also in many common gadgets that we often use. Nowadays, pocket cameras detect faces in order to properly set up the scene (e.g., focus on the face) and smartphones often have capability of running augmented reality software for telling its users what they are pointing at with their smartphone's camera.

Face recognition criticism

Face recognition itself is very useful and also very "cool", but it has some big drawbacks too. In big cities, people are watched almost everywhere — in markets, museums, on traffic lights, public transport; this list could go on and on. For example, in China the expenses of video surveillance are expected to reach $79 billion in 2015, though the capital of video surveillance is still Great Britain with around 3 million cameras [12]. Some people would say "I do not do anything illegal, so why should I care", but for others the idea of being watched most of the time is very uneasy. The amount of materials taken from surveillance cameras is enormous so it is almost impossible to process it manually. Without doubt, implementation of face recognition in video surveillance will shrink our privacy even more. And it has happened already. "A jail in Alabama uses it to check those leaving against prisoner records. Mexican prisons use it to identify visitors" [12]. Such use is understandable, but recognizing people on the streets does not make that much sense. In addition, the results are far from satisfactory: "Critics of the technology complain that the London Borough of Newham scheme has, as of 2004, never recognized a single criminal, despite several criminals in the system's database living in the Borough and the system having been running for several years." [13]. This could be also be an issue of fundamental democratic principles, because "freedom of speech is reduced, when mere physical attendance

at protests goes on record"[12].

Social networks

Another field of face recognition application is photo annotation. Imagine we have a large database of photos and we want to search for photos with certain people on them — we do not want to go through the whole set each time, and so it is essential to have such database annotated. The obvious approach is letting the user annotate it first, but it is not very comfortable.

So computer photo galleries like Google Picasa started to search for faces on background and offering the user faces to tag. Once a face of a person is tagged the other instances of the same face are tagged automatically. A logical step was to broaden this function to web-based galleries, like Google's Picasa Web Albums. These albums are on the web, but the set of people is still unique for each user.

Where people tagging becomes really interesting is in social networks. There you do not have just a name matched to the face, but the whole digital profile, which can expose quite a lot in case of some people. About a year ago, Facebook launched face recognition to help users tag photos; Google+ followed half a year or so later.

These social networks offer privacy settings (e.g., user can select whether his face can be automatically offered for tagging), but do not provide any settings concerning the recognition process. We have implemented a web application which has a similar environment to the one of social networks, but the user can influence the recognition process. Albums and people are currently not shared between users, but it would not be difficult to do so. As explained in the Method chapter, the success rate of recognition depends heavily on the fact whether recognized face was in the initial set. Therefore, Facebook and other social networks will therefore have much better results, because their databases are incomparable with the database of photos common users have.

Thesis structure

The following text is divided into chapters. The first chapter describes the method used for face recognition. It is divided into three sections describing the three main components of the recognition process. The second chapter describes our implementation including programmer and user documentation. The third chapter includes results of some experiments and explains the default settings based on these results.

1. Method

The method consists of three stages. First stage is *face detection* where areas in the image where face is are discovered. When faces are found they have to be further processed. This process is called *normalization*, because the attempt is to have similar lighting, face posing and so on on all faces prior to recognition. Recognition itself is the last step.

1.1 Face Detection

Detection is the first part (excluding some preprocessing) of automated face recognition process. It may be the most important phase in a machine aided or perhaps fully automated photo annotation system, because selecting regions with faces is much more exhausting for the user than telling the system whose detected face it actually is. Fortunately, face detection is nowadays considered mastered.

In our application we use detection system introduced by Paul Viola and Michael J. Jones in 2001 [1]. This system uses cascaded feature classification that allows very fast as well as efficient object detection. The authors presented this system for face recognition, but it is general and we will use it for eyes, mouth and nose detection as well.

The principle is to search for certain features in the input image. Many weak classifiers are combined to form a strong one. Organising them into a so-called cascade along with handy image representation together achieve its high speed of detection. We will now describe this technique. If the reader is interested in further details, may he read the original paper.

1.1.1 Features

The detection system does not work directly with the value of pixels, but with the simple features. The value of a feature is computed by summing up the value in certain adjacent rectangles and subtracting them from each other; if the value exceeds a certain threshold the feature is said to be present. We use three kinds of features, shown in figure 1.1:

- two rectangles, adjacent horizontally or vertically — shown as A and B
- three rectangles, adjacent horizontally — C
- four rectangles forming another one — D

Value of the feature is the difference between black and white regions.

1.1.2 Integral image

In order to compute these features rapidly, a so-called integral image is used.

Definition. *An Integral image is a matrix where every element* ii *is the sum of all pixels* i *to the left and above* ii *in the original image.*

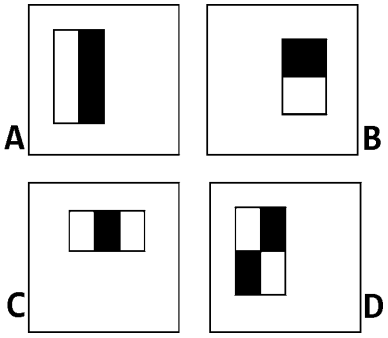


Figure 1.1: Feature types

$$ii(x,y) = \sum_{\substack{0 \le x' \le x \\ 0 \le y' \le y}} i(x',y')$$

The advantage of such image representation is that the computational time of any rectangular sum is constant: we only need three integer operations at most. As seen in figure 1.2, if we want to compute sum in area D we compute $4-3-2+1$; for area C it is $3-1$ and so on.

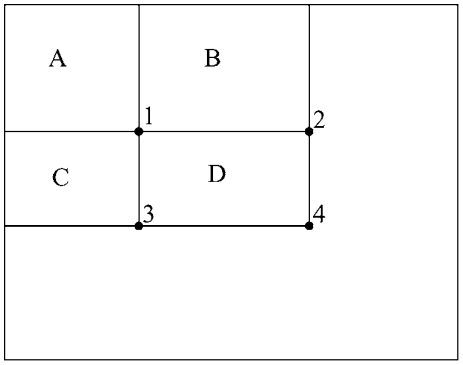


Figure 1.2: Integral image

1.1.3 Cascade classifiers

As the reader may have suspected, searching for features (which actually are just differences between pixel values of image areas) and deciding whether image window[1] is face or not based on their presence or absence, isn't quite precise. In fact, it is not much more reliable that random guessing. This feature-based classifier is *weak*.

So why is the described method good? The answer is *classifier boosting*. The idea is to combine many weak classifiers that stand together as a strong one. We will form the classifiers into a cascade.

Definition. *A cascade classifier is a set of* n *classifiers* $c_1, ..., c_n$ *which accepts the input if all of its subclassifiers accept it. Cascade classifier rejects the input when any of the subclassifiers rejects it. Cascade classifier works sequentially -* c_1

[1] cut-outs of an image are scanned, so the general scene can be detected

is the first to evaluate the input, positive result triggers evaluation of c_2 *and so on;* c_n *evaluates it at last. If any of the classifiers has a negative result, the cascade is stopped.*

The improvement in speed is obvious: only the promising parts of an input image are tested thoroughly, and non-faces are quickly rejected. The order of subclassifiers is very important: at the beginning of the cascade there have to be simple and fast classifiers which reject the majority of false samples in order to prevent useless operations on more complex — and slower — following classifiers.

Improvement in accuracy is not so straightforward. Complicated classifiers, which use many features and are used later in a chain of subclassifiers tend to have more false positive results. In a cascaded classifier it is not such a big problem, because images which do not represent face but normally would be marked as positive are rejected by some simpler classifier earlier in the process and do not arrive in this complicated one at all.

Figure 1.3: Possible beginning feature detector

Example of feature detector at the begin of a cascade is shown in figure 1.3. This detector yields a fact that region of eyes is often darker than region of nose and cheeks and that nose bridge is brighter than eyes.

1.1.4 Classifier training

Classifiers must be trained on a set of positive and negative sample images, but this is out of the scope of this work, so we use classifiers provided with OpenCV.

1.1.5 Examples

Figure 1.4: Example of face detection. On the right image face was not detected, probably because of the shadow covering parts of the face.

1.2 Face Normalization

Eigenface-based face recognition system — the one that we use and that is described in the next chapter — is very sensitive to the state in which it processes faces. The success of such recognition depends heavily on similar light conditions, face position etc. of the faces. We will now introduce the sequence of steps taken to obtain a *normalized image* from the image that we get from the face detector.

1.2.1 Image straightening

The first task is to straighten the face. To accomplish this, we look for important face parts and use similar detector as described in the previous chapter. We try to locate the eyes. Once they are found, process is quite straightforward. This case is shown in figure 1.5.

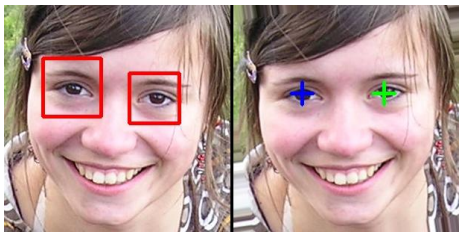


Figure 1.5: Eyes base image straightening. The left image shows eyes detected by the feature detector and the right image shows the straightened image with estimated eyes position.

If eyes can not be found (they are closed, person is wearing sunglasses, or there are just poor light conditions), we try to locate the mouth and the nose. From their position, we try to guess [2] the position of the eyes and then straighten the image as in the previous case.

If neither nose nor mouth cannot be found the image is not straightened and is said that it is probably not face. This eliminates false positives detections as well as some correct detections. It is not ideal, but faces where important parts can't be found won't be properly recognized[3] either, so it does not matter so much. An example of a better scenario is shown on figure 1.6

1.2.2 Image resizing

Of course we have to recognize images with the same dimensions it is also essential that faces are positioned correctly. Our system therefore fixes the size of a person's eyes and the resizing is based on that. After the image has been resized and positioned the outer parts are cropped out. What can happen is that resized image — the person's face — is smaller than desired image size. In that case, the edge of an image is repeated until the image has the desired size. We do not use black borders because it would mess up histogram equalization which is used later.

[2] we multiply the distance between mouth and nose by (user defined) constant to get the eyes-nose distance, the distance between eyes is computed similarly; it is obvious that result can not be satisfactory for all faces

[3] people must face the camera, otherwise this method will not work

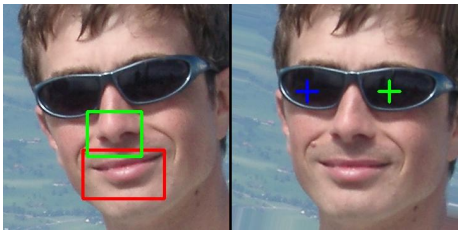


Figure 1.6: Mouth-nose based image straightening. The left image shows eyes detected by feature detector and the right image shows the straightened image with estimated eyes position.

Figure 1.7: Desired face height is greater than the actual photo. Note the bottom line being repeated.

1.2.3 Light compensation

A photo can be taken under very different light conditions — direct sunlight, indoors with artificial light, etc. What can happen is that part of the face has lower light than other parts (e.g., is shadowed by nose). We try to reduce such local differences by an algorithm called *Single Scale Retinex* [2]. Also, the image is converted to grayscale prior to retinex processing, because one-channel images are used for face recognition.

Definition. *A Single Scale Retinex Works as follows:*

$$R(x,y) = \log I(x,y) - \log\left[F(x,y) \otimes I(x,y)\right]$$

Where $R(x,y)$ *is the output image,* $I(x,y)$ *is an input image,* $\otimes$ *is convolution operator and*

$$F(x,y) = Ke^{-\frac{r^2}{c^2}}$$

Where $r^2 = x^2 + y^2$, c *is Gaussian surround constant and* K *is selected so that*

$$\iint F(x,y)\,\mathrm{d}x\,\mathrm{d}y = 1$$

This definition looks complicated, but in fact it is not so difficult. We compute Gaussian blur of an input image. Thus, we divide each pixel from original image with the value of the corresponding pixel in the blurred image. The last step is normalizing the values to fit the interval $[0, 255]$. Figure 1.8 shows some of the results.

Figure 1.8: Retinex image processing. Leftmost is the straightened image, follows grayscaled and rightmost is retinex processed.

1.2.4 Histogram equalization

Single-scale retinex will correct local light differences. But we have to correct them also globally — it would be difficult to recognize images if some were brighter than others. That is where *Histogram equalization* comes to the scene.

Histogram equalization transforms the image such that the *Cumulative distribution function* is linear and that its derivative is greater than zero. The computation [3] is not difficult:

1. Compute histogram H for src where src is the source image
2. Normalize the histogram so its sum fits in 255
3. Compute the integral of the histogram:

$$H'_i = \sum_{0 \le j < i} \mathrm{H}(j)$$

4. Transform the image using H': $dst(x, y) = \mathrm{H}'(src(x, y))$ (dst is the destination image)

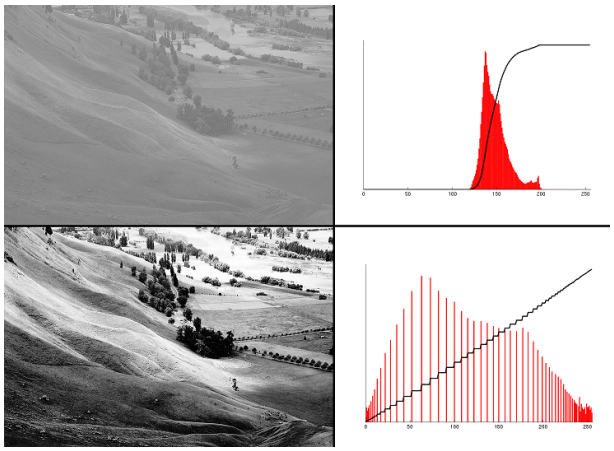


Figure 1.9: Input and output of histogram equalizatino process. On the right, there are corresponding histograms (red) and cumulative distribution function (black). The images are taken from Wikipedia [4]

1.2.5 Gaussian Blur

Last step of face normalization is to blur image a little bit, so that the minimal differences or picture errors won't matter.

Definition. *Gaussian distribution in two dimensions:*

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

We create a matrix which has elements of values corresponding to normal Gaussian distribution in two-dimensional space. Such matrix is of type $n \times n$ where n is odd number.

This matrix will be used as a convolution kernel, which means "Each pixel's new value is set to a weighted average of that pixel's neighborhood. The original pixel's value receives the heaviest weight (having the highest Gaussian value) and neighboring pixels receive smaller weights as their distance to the original pixel increases. This results in a blur that preserves boundaries and edges better than other" [5].

Result of blurring image this way shows figure 1.10

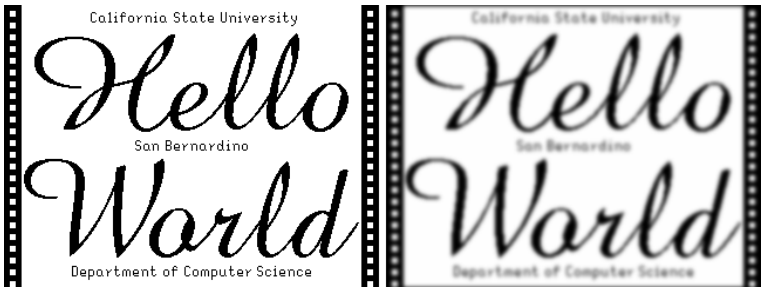


Figure 1.10: Normal image (left) and blurred image (right). Images are taken from Wikipedia [5]

1.2.6 Overall example

Figure 1.11 shows all steps of face normalization progress. Going left to right then up to down:

- input image
- eyes are detected
- face is straighten and eyes aligned
- face is resized
- converted to grayscale
- Single-scale retinex is applied
- histogram is equalized
- image is blurred

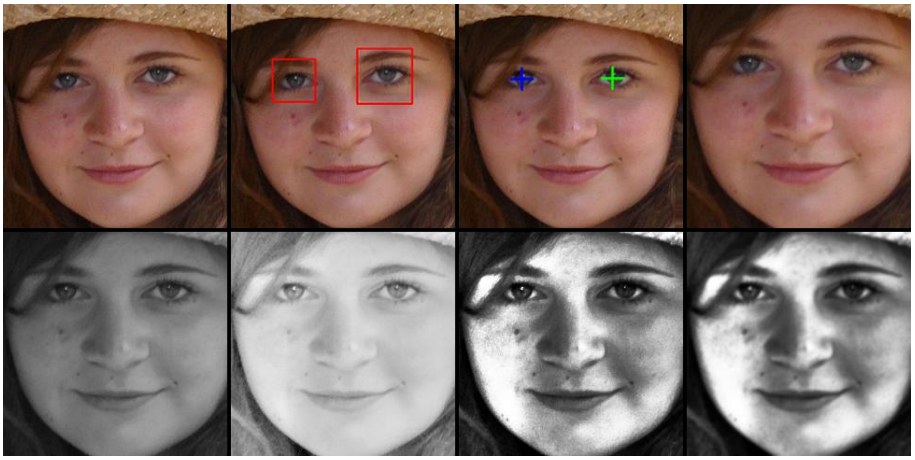


Figure 1.11: Overall progress of face normalization.

1.3 Face Recognition

Here we will describe the final step of the method implemented, face recognition itself. We use an approach that was originally introduced by Matthew Turk and Alex Pentland [6].

1.3.1 Image as a point

We perform face recognition on gray-scale images, so our image is a matrix such that each point has a value from 0 to 255. But we will not be looking at an image as a set of vectors (a matrix), but as one vector, formed consecutively from the image vectors. Here is an example:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

$$\mathbf{a}_{m,n} = \left(a_{1,1}, a_{1,2}, \cdots, a_{1,n}, a_{2,1}, a_{2,2}, \cdots, a_{2,n}, \cdots, a_{m,1}, a_{m,2}, \cdots, a_{m,n}\right)$$

A is an input image and $\mathbf{b}$ is of course the corresponding image vector.

Notice that the n-dimensional vector is the same thing as a point in an n-dimensional space. Image represented as a point is very useful, because we can compute difference between images as a distance between points in space (Euclidian, Mahalanobis, Manhattan, ...).

But computing it this way will not yield good results because any minor difference in the pixels' value will have great impact on the computed distance. We will therefore perform dimensionality reduction such that distance is changed significantly only if there is a major difference between the images.

1.3.2 Principal Component Analysis

> "PCA is mathematically defined as an orthogonal linear transformation which transforms the data to a new coordinate system such that

the greatest variance by any projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on." [8]

In other words, PCA looks for a set of orthonormal vectors that best describes the distribution of the data [6]. The input data we will call a *training set*.

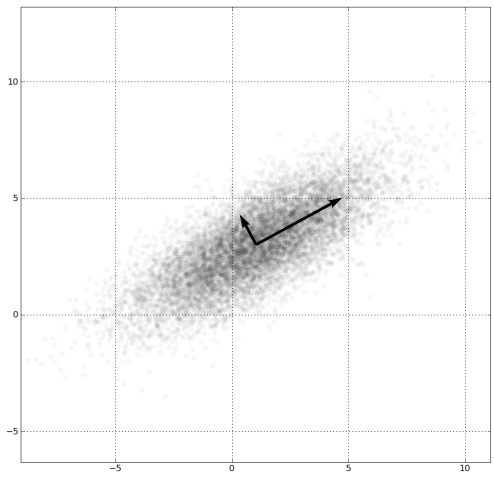


Figure 1.12: Example of PCA. Dataset is a multivariate Gaussian distribution (shifted and rotated), vectors indicated are the new basis, computed by PCA — the output of PCA

Definition. *The covariance between two jointly distributed real-valued random variables X and Y with finite second moments is*

$$\operatorname{Cov}(X,Y) = \operatorname{E}\left[(X - \operatorname{E}[X])(Y - \operatorname{E}[Y])\right]$$

[9] where E *is* Expectation [4] *and* X, Y *are vectors representing random values.*

Definition. Covariance matrix C *is a matrix such that*

$$c_{ij} = \operatorname{Cov}(\mathbf{X}_i, \mathbf{X}_j)$$

where $\mathbf{X}_i$ *and* $\mathbf{X}_j$ *are i-th and j-th vectors from a dataset.*

As we said earlier, the output of PCA is an orthogonal set of vectors and their directions reflect variety of dataset. *Such vectors are the eigenvectors of the covariance matrix.* The first principal component is the eigenvector corresponding to the greatest eigenvalue, second is the one with second greatest eigenvalue and so on.

1.3.3 Eigenfaces

Regarding the principle of representing images as vectors, we will do the exact opposite. We will visualize basis vectors given by the PCA, the result is ghostly-looking "faces" — *eigenfaces*. The cause of such ghost look is the fact, that every eigenface is a compound of faces given to the PCA in the first place.

[4] $\operatorname{E}(\mathbf{X}) = \sum_{i=1}^{n} p_i \cdot x_i$, $\mathbf{p}$ is probability of $\mathbf{x}$

Figure 1.13: First ten eigenfaces computed by PCA

1.3.4 Using eigenfaces for face classifying

As we have already said, PCA computes new basis vectors. So if we want to classify a new face, we will just project it onto PCA-computed space, that will give us a vector — or a point in multidimensional space — and we will call that a *face descriptor*.

Another point of view is that we construct a linear-combination of eigenfaces that forms the original face.

$$F = \sum_{i=1}^{n} \mathbf{b}_i \cdot \mathbf{e}_i$$

F is the original face n is number of eigenfaces, $\mathbf{e}_i$ is the i-th eigenface and $\mathbf{b}_i$ is the coefficient, which is i-th coordinate of face vector projected to new PCA space.

1.3.5 Further dimension reduction

The first principal component carries the most useful information, the second one carries less, and so on. The question which will be subject of experiments in later sections is whether we need all of the principal components or not, because the latter ones are mostly noise. The answer will probably be that we do not need all of them.

Figure 1.14: Face reconstruction from eigenfaces: on the left is the original image, center image is the one reconstructed from all computed eigenfaces (18), the right image is reconstructed just from 10 eigenfaces.

1.3.6 Comparing descriptors

Now, if we want to decide whether faces are alike, we can compute the distance between their descriptors. If it is below a certain threshold, we say that they belong to the same person, otherwise we establish that they do not.

Of course, the success of such comparison depends heavily on the fact whether the compared faces were in the initial face set on which the PCA was performed, because otherwise face can not be properly projected onto the PCA subspace.

2. Program

In the first part of this chapter we will describe implementation of the method described in the chapter preceding. The second part is user documentation.

2.1 Architecture

From the very start of this project, emphasis was put on separation of its components. The core is a dynamic-linked library, which does most of the work (and the most interesting one). This library is written in C++ and is distributed in both 32-bit and 64-bit versions. Because it is DLL, it is not so widely portable — it will only work on Windows machines. The library is not intended to be used directly: for the programmer comfort, the .NET wrapper is provided. This wrapper does what one can expect from such a thing: it maps between managed (.NET) and unmanaged types and takes care of cleaning memory used by the library. The task of recognizing multiple photos is ideal for parallel computing. Somewhere the work is divided by the wrapper, somewhere by the library.

Two GUIs were created: desktop and web GUI. The desktop GUI was planned from the beginning: something to present the possibilities of the library. Though web GUI was added while application developed, it has become the major one, leaving the desktop GUI just being a testing tool.

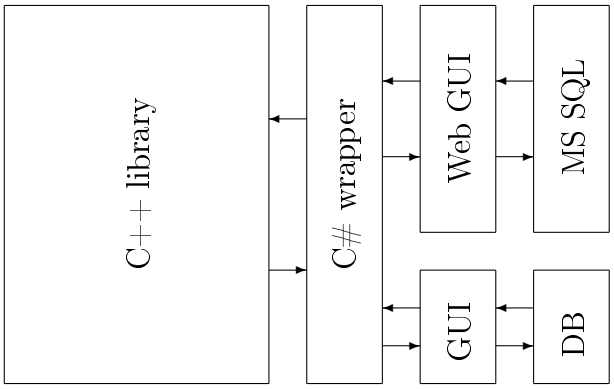


Figure 2.1: Architecture. Arrows visualize communications between components.

2.1.1 Library

The library is written in C++ and compiled natively for x86 and x64 processors, for the sake of speed. Managed code has spread extensively in last few years and its use has extended even to large information systems or video games. There are even some reasons for managed code to be faster than the unmanaged[1], but our library works with large amounts of data, that cause many memory allocations and they would result in many *garbage collections* when managed-coded, and so unmanaged code is our choice.

[1] JIT compiling, fast heap allocation (.NET allocates space only at heap "top", therefore at constant time — opposite to C++ *new* operator that searches for empty space in linear time), detailed knowledge about machine used etc.

We use *OpenCV* and *Threading Building Blocks* libraries in this project. Through all our native code we use OpenCV types and OpenCV is in charge of the first part of the face recognition process — face detection. Intel's TBB is a great library for code parallelization and we are using it when creating eigenfaces database (the training set of faces). Input for eigenfaces database creation is set of photos, so on each photo faces are firstly detected and normalized afterwards; all this can be done parallel — one thread for one photo, for example, and that in fact is how we do it. So the most exhausting part is parallel, but the PCA decomposition itself is serial. I do not know about any method for solving this problem on parallel and I honestly believe that the result of such thing would not be worth the effort.

The library is in the form of DLL and exports the required functions. These functions are imported by the wrapper which can be comfortably linked by any .NET application.

The library also handles persisting some data, specifically eigenfaces database and database of people descriptors. That is because this way C++ can directly work with the required data and is not dependable on third-party database. Also, the interface is thinner and the choice of database is on the application using our library. So, when asked "who is on this picture?" the library responds just with an integer that represents ID of person. Other functions works similarly, such as "whose face is the one not recognized" and so on. Other details are stored in a separate database and are totally independent of the library.

There are three main objects that form the library: *Detector*, *FaceNormalizer* and *FaceRecognizer*. Auxiliary class is *ImageProcessor*. Each of these classes has a minimum of public methods and their communication is pretty straightforward. For example when *FaceRecognizer* gets the task to recognize people on multiple photos, it calls the method *GetDetectedObjectsImages*[2] on *Detector* for each photo, and subsequently it calls the method *Normalize* on *Normalizer* for each image returned by *Detector*, and that is all for communication.

There are also some *structs* used to carry information: such as numerous settings, bitmaps returned in exported functions and so on.

Draft of library classes and structures is shown on figure 2.2:

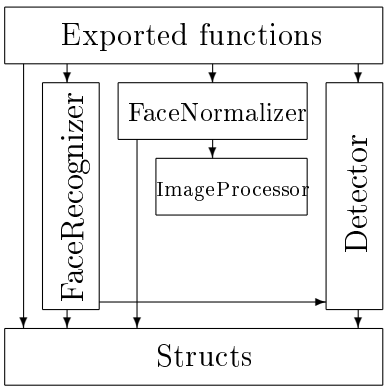


Figure 2.2: Library architecture. Arrow from A to B means that A uses B.

[2]if someone wants just detect where faces are, he might call *GetDetectedObjectsRectangles*

2.1.2 Wrapper

Wrapper structure copies the structure of the library. It is after all the function of wrapper just to "wrap up" some library and expose its functions in a more comfortable form. So you can instantiate practically the same classes in wrapper as you can in the library.

The only interesting part of wrapper is parallelism. If the task of recognizing faces on multiple photos is given, that set of photos is divided and distributed fairly to all available processors. So the library gets a serial task, and parallelism is provided by the wrapper.

2.1.3 Desktop GUI

There is not much that is interesting from the programmer point of view in the desktop GUI. It just demonstrates the library possibilities and will be described in the *user documentation*. It is .NET Windows Forms application.

2.1.4 Web GUI

Web GUI uses ASP.NET technology, so it can comfortably use .NET wrapper described earlier. The web application supports multiple users, each user can add his own albums, have his unique settings for detection, normalization and recognition etc. User accounts management uses *Altairis Web Security Toolkit*.

It uses MSSQL database for storing all informations about persons — the library, as said before, is working only with IDs.

A major challenge was handling web requests that could come anytime, the resulting parallelism, and progress indication[3]. While photos in album are being recognized, the progress bar is shown and together with different colors of photos borders indicates the whole progress. This is powered by AJAX: the whole page does not need to be refreshed — only javascript code asks the server what progress percentage is and what photos are processed already. The C++ library does not have the possibility to indicate progress, so the library gets only one photo to process in batch and it is immediately stored in database. Progress indication is therefore provided only by a database query. Parallelism is preserved, though realized on higher level: by application.

I wanted to avoid implementation of a web photo gallery, so our application uses other ones. When the user wants to add an album, he has to provide URL of a web-based gallery — Picasa Web Albums, Rajče or Facebook. Both Picasa Web Albums and Rajče offer RSS channel that can be easily parsed. Facebook does not have such a thing, so its album page source code is parsed, which has many disadvantages: if somebody on Facebook decides to change their code our import ceases to work, such parsing is slow and so on. Our application stores links to images in those galleries and works with them. Most of the time, it is sufficient, because the web browser does not care from where it downloads data.

But if we want to recognize photos, they have to be on the server. So they are downloaded to user and album specific temporary folder and processed afterwards. These folders are cleaned when the pictures are not needed.

[3]This is a problem for web application, because its client-server architecture is not intended for events raised by the server.

2.2 User documentation

2.2.1 Web application

The application is available on `http://siret.ms.mff.cuni.cz/facereco/`.

Requirements

The application is HTML5 valid so it will work in any HTML5 browser, which all modern browsers are. Thanks to HTML5 backward compatibility, it will work on almost all browsers. AJAX and therefore javascript is used, though not required. If the browser does not support these technologies, the application will work, just will be less comfortable. The application was tested and major functions work even on unusual devices such as the Amazon Kindle e-book reader.

Registration

For the application to be customizable, user accounts are used. On the main page, there is a link to registration. After the registration is complete, the user can add his own albums, settings, database of people, database of eigenfaces — all this is user-dependent.

Themes

In the upper right corner, the user can choose between *dark* and *light* theme.

Albums

After successful registration, next thing is to add some photos. On the *Albums* page, there is a link *Add album*. User is now given a choice of three web galleries to add photos from - *Picasa*, *Rajče* and *Facebook*. Also, he can add a testing database from California Institute of Technology.

In the case of Picasa or Rajče, the user will be asked to enter URL of an album. When adding from Facebook, he has to provide the source code of the webpage.

Album can be deleted by clicking the 'x' symbol, which appears when the mouse cursor hits its area. After clicking the album's name or thumbnail, its contents will be displayed.

Recognition

In album, click "Recognize!". The album starts being processed and a progress bar indicating how much work is already done is shown.

Every image has a border - if it is grey, the image has not been recognized yet. If it is blue or orange[4], it has been recognized. Some processed images will have a number in the upper-left corner; this indicates a number of faces detected on this image.

By clicking the image, the user is brought to a new page with that image displayed. Next to it, there is a list of people recognized. When clicked there (or

[4] depending on selected theme

a face in the image) the user will have an opportunity to identify the person if it was not recognized properly or was not recognized at all.

The application is learning on-the-fly with every new face added to database. So the more photos the user will processes and the more people tags, the better the application will be. When choosing persons in database the one on the photo, they are sorted by distance from the input face. Therefore, the most probable faces are at the beginning of the list.

If the image is of poor quality, there is unusual lighting or some other conditions are bad, the faces will probably not be detected. To solve this, there is a button "Add new face". Then the user can specify an area in image that will be looked on as a face. But note that the conditions due to which the face was not detected in the first place are still bad, so recognition will not be accurate.

People

Once the user tags some people, they are listed on the *People* page. From there, the user can change person's name, profile photo and view albums and images where the person is.

Figure 2.3: Web application during recognition process. Grey-bordered photos are not processed yet, opposite to orange-bordered ones.

2.2.2 Settings

Both web and desktop GUIs have multiple settings: now we will try to explain their meaning.

Detector settings

- *Cascade*: here you choose which cascade will be used in detection
- *Minimum of neighbors* is a number indicating how many other candidates have to be near the one to be detected as a face. The higher number, the less detections yo will get.
- *Scale factor* is a number depicting how the testing rectangle is growing. For example: if the value is 1.1, the image is once searched for rectangles of volume n and then for rectangles of volume 10% greater. The higher the number, the less detections you will get.
- *Rotation steps*: this is useful for situations, where people are not standing on a photograph so their faces are oriented elsewhere. Full circle is divided into n parts, where n is the rotation steps. Then the image is processed, then rotated, then processed again, ... If n is 1, the image is processed just once. If n is 10, the image is rotated every time by 36 degrees. Be aware, that face detection is by far the most exacting task, and so 10 rotation steps will make the whole process 10 times slower than with just one step.
- *Maximum image size*: For performance reasons, images are scaled down before performing detection. Here you can specify maximum length of any image's edge.
- *Minimum object width scale*: Too small objects are ignored. Here you can specify the minimum value of object width for being tested. It is a value relative to the image width.
- *Minimum object height scale*: Similar to above.

Normalizer settings

- *Face width*: Desired normalized face width[5]
- *Face height*: Desired normalized face height.
- *Eyes position - X*: Desired distance of eyes from corresponding edge.
- *Eyes position - Y*: Desired distance of eyes from upper edge.

  Sometimes, eyes can't be detected. If person is wearing sunglasses, for instance. Then eyes are aligned based on the distance between mouth and nose. This is not ideal, of course.

[5]Face width and face height can be adjusted only from desktop application when creating eigenfaces. When using application, these values are used no matter how it got changed in GUI, it is stored along with eigenfaces.

- *Half eyes distance*: value indicating, how many times is half of the distance between eyes greater than the distance between person's mouth and nose.
- *Eyes distance from mouth*: similar value, now about vertical distance.
- *Retinex-surround constant*: size of the surrounding, which is relevant when equalizing lighting.
- *Blur-surround constant*: how much the image is blurred.

  Then there are settings for detecting face features. They are similar to the detector settings. Note that eyes are searched three times. If one isn't successful, another search follows.

Recognizer settings

- *Eigenface number*: number of principal components to use.
- *Recognize distance*: is a threshold value, for face being recognized. If the nearest person is closer to processed face, it is marked as this person. If it is more distant, it is marked as unknown.
- *Distance Metric*: is metric used to calculate distance between faces. User can select between *Mahalanobis*, *Eucleidian* and *Manhattan* distance.

  In addition, the web application has a possibility to create eigenface database from all pictures belonging to the user; the desktop application, on the other hand, can create it from the photos in a certain folder, selected after clicking on *eigenfaces* button on *Options and Testing* tab.

2.2.3 Desktop application

Desktop application is divided into five *tabs*.

- *Photo browsing* tab is for selecting which photos will be recognized. In the left column there is directory tree, on the right there are displayed images from selected directory. The user can check images that will be recognized by clicking the *recognize!* button or will have faces detected by clicking the *detect!* button. Options for selecting all images, none image and to invert selection are provided.
- *Recognized people* is the tab to which user is brought after recognition ends; it browses through recognized photos, allowing the user to tag photos that are not matched to people in database and to correct mistakes.
- *People in the database* is a tab where *reconstructed*[6] faces of people are shown.
- *Training database – Eigenfaces*: first face here is *average face* — face-like image computed as an arithmetic mean of all faces in the training set, others are computed eigenfaces (principal components), sorted by corresponding eigenvalues

[6] back projected from PCA subspace

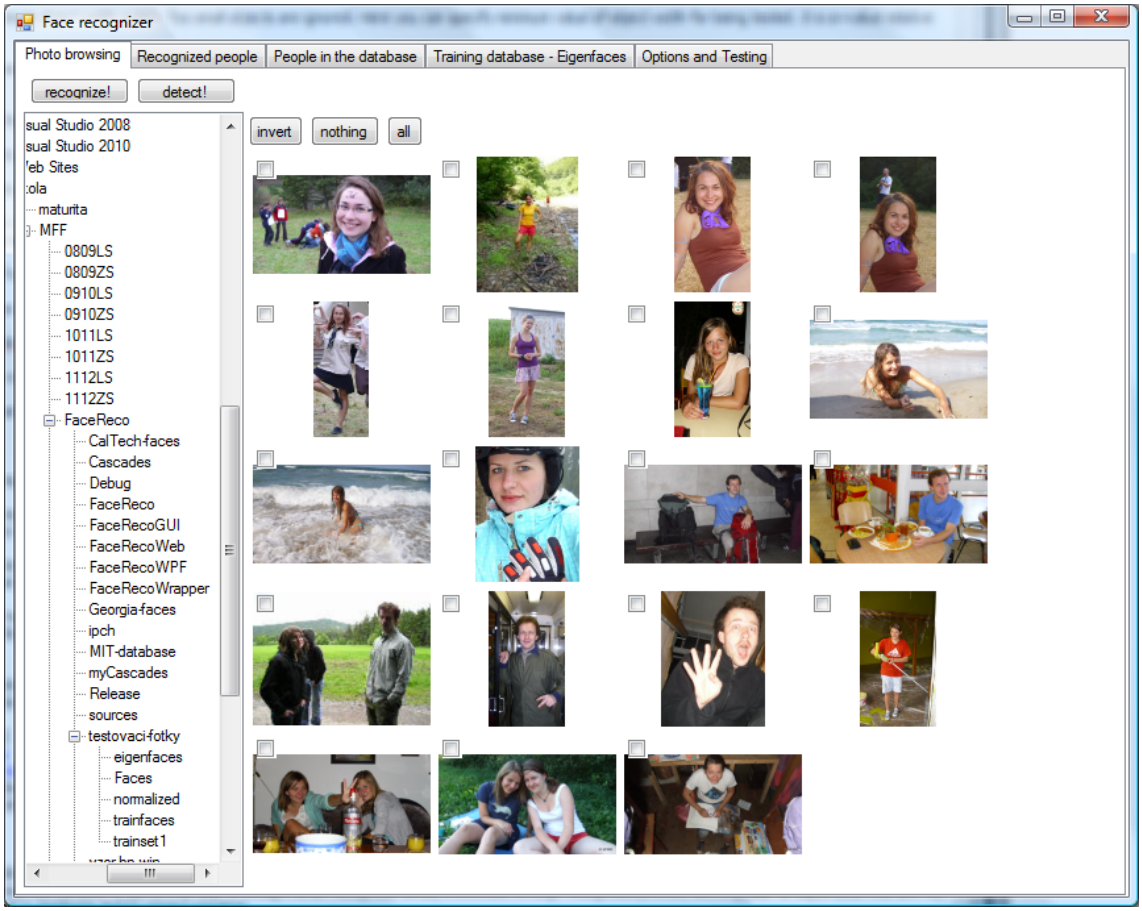


Figure 2.4: Desktop application.

- *Options and Testing* tab is where settings can be adjusted. Settings substance is the same as in the web-application, but here you can also choose where to store eigenface database and database of people descriptors. Also, you can try *face detection* on common photo and *face normalization* on picture of a face.

  But most important function here is creating the database of eigenfaces. When the user clicks button *eigenfaces*, he will be asked to pick a directory. From pictures contained in that directory faces will be extracted and used to create eigenfaces.

2.2.4 Use of a wrapper

Wrapper has different versions for x86 and x64 processors. It is not compiled as .NET's *Any CPU*, because that way the corresponding (x86 or x64) native libraries could not be determined. All DLLs must be in the same folder (or perhaps somewhere else after appropriately editing system's PATH variable, but I haven't tested such solution).

After linked to some .NET application, wrapper provides the functions described in the Architecture section.

3. Experimental results

Face detection, normalization and recognition has many parameters, their meanings were explained in the previous chapter. But what values should be used? This chapter tries to give the answer.

3.1 Face detection

Many trained cascade classifiers comes with OpenCV. We have not trained our own ones, so the only relevant parameter for testing our classifiers is *minimum number of neighboring faces candidate*. Other parameters, like *minimum object size*, *maximum image size* and *scale factor* can increase detection speed and decrease detection accuracy — we set them so they won't affect the result set.

We used set of 33 photos. There is a variety in test images: in number of people on photo, relative size of face, lighting and so on.

Following four tables corresponds to four OpenCV classifier cascades. Each table exposes *minimum neighbors* (the parameter), *correct detections* (detected faces), *false positives* (detections where there is no face in fact) and *undetected* (missed faces). See tables 3.1, 3.2, 3.3 and 3.4.

minimum neighbors	correct detections	false positives	undetected
0	33	388	23
1	48	64	7
2	**51**	**34**	**8**
3	46	18	12
4	47	14	11
5	44	12	13
6	43	7	14
7	42	5	15

Table 3.1: Cascade frontal_face_alt

Table 3.2: Cascade frontal_face_alt2

minimum neighbors	correct detections	false positives	undetected
0	22	632	34
1	50	138	3
2	48	68	9
3	48	35	9
4	49	18	7
5	44	17	9
6	48	9	8
7	43	8	10

As mentioned in the face normalization part of Method section, in normalization process the image can be also rejected as non-face. Three times eyes are tried to be detected and when it is not successful mouth and nose is searched. If

Table 3.3: Cascade frontal_face_alt_tree

minimum neighbors	correct detections	false positives	undetected
0	46	124	14
1	44	30	14
2	38	13	15
3	38	7	18
4	38	3	19
5	37	2	20
6	36	1	21
7	36	1	21

Table 3.4: Cascade frontal_face_default

minimum neighbors	correct detections	false positives	undetected
0	25	867	33
1	52	306	6
2	51	169	8
3	48	105	9
4	48	67	10
5	46	49	12
6	46	39	12
7	46	28	13

these parts can not be located, image is rejected. So, *frontal_face_alt* cascade with minimum number of neighbors of 2 is selected as the default face detector, because it has most correct detections while number of false-positives is still reasonable[1].

Attempt is to detect as most faces as possible, leaving false-positives to be rejected by the normalizer. If somebody would want to use just the face detector, he might want to select different settings with lower number of false-positives.

As *minimum neighbors* increases, false positives decrease together with increasing number of undetected faces. From some point, there is no sense in other measurements with higher parameter, so there we stopped our experiments.

So what settings use for detecting face parts? Here we can't be so benevolent with false positives. Following five tables, each for one classifier, show results from experiments similar to the face detection one. Bold rows are selected as the defaults.

Test set counted 42 faces, mostly detections from the previous experiment with some faces added later. See tables 3.5, 3.6, 3.7, 3.8 and 3.9.

The total success (that means passing initial detection and all stages of normalization) of detection is 76% with no false positives. Some photos used were taken with low resolution, somewhere people are not precisely facing the camera. With more carefully taken photos, the success would definitely be better.

[1] This work is about *frontal face images*. But some classifiers can detect even faces somewhere in the middle between "profile" and "front". This is not bad, but it also not expected, so there were cases that I counted as *correct* if detected but not as *undetected* if not. **This is the reason why *correct detections* + *undetected* is not invariant**.

Table 3.5: Cascade eye (used for first time eyes detection)

minimum neighbors	correct detections	false positives	undetected
0	73	52	13
1	68	11	18
2	66	8	18
3	63	4	21
4	60	4	24
5	57	3	27
6	55	3	29
7	55	1	29
8	55	1	27
9	53	1	27
20	**53**	**0**	**31**

Table 3.6: Cascade mcs_left_eye (used for second time eyes detection)

minimum neighbors	correct detections	false positives	undetected
20	**57**	**12**	**27**
25	51	8	33
30	49	7	35
35	48	7	36
40	45	5	39
45	45	4	40
50	43	3	41

Table 3.7: Cascade mcs_right_eye (used for third time eyes detection)

minimum neighbors	correct detections	false positives	undetected
0	74	33	10
1	71	21	13
2	**66**	**16**	**16**
3	67	14	17
4	65	9	17
5	66	9	18
6	64	8	20
7	66	5	20

3.2 Face recognition

There are many parameters in face recognition that can be adjusted. Some are easy to evaluate subjectively, like *retinex surround constant* that affects light normalization — it is observable. But probably most important is the choice what part of face is used for recognition (e.g., whole head including ears and haircut or just an area from eyes to mouth). In our application, we can position eyes on exact coordinates in the face cut-out and this way determine recognizable area. This parameters are what we will discuss.

We took 10 subjects for our experiments. First, we created descriptor of one

Table 3.8: Cascade mcs_mouth

minimum neighbors	correct detections	false positives	undetected
5	39	72	3
10	39	54	3
15	38	43	4
20	38	36	5
25	36	30	5
30	32	28	10
35	32	24	11
40	**32**	**23**	**11**
45	31	19	11

Table 3.9: Cascade mcs_nose

minimum neighbors	correct detections	false positives	undetected
5	38	17	4
10	36	4	6
15	35	1	7
20	34	1	8
25	32	0	10
30	**33**	**0**	**10**
35	29	0	13

photo of the subject. Next we created descriptors of the rest of the subject's photos and calculated eucleidean distance between them. We computed median from all these distances[2] for each subject; therefore we computed mean of these medians. This procedure gives *mean distance* that characterizes used parameters.

Also, we counted *mistakes* — how many times face of another person was marked as the nearest one. Table 3.10 shows medians of distances based on eyes location (23, 30 or 40 pixels from borders), 120x120 pixel images were used. There is also shown mean of that medians and number of mistakes. It is notable, that the best results are obtained when eyes are shifted 30 pixels from image borders, such face shows figure 3.1

The other important parameter is a number of principal components which are taken as a basis. If we took all of them, even image noise would have notable impact on recognition, if we took small number, faces can't be properly distinguished (and restored, if needed). We performed several experiments using the same method as earlier — counting mean distance and mistakes. Because the mean distance in lower dimensional space distance is of course shorter than in space with more dimensions, mean distances were normalized to values between 0 and 100.

[2]median is better than arithmetic mean, because it is robust — one photo can be taken in difficult conditions and therefore be very distant from others

Table 3.10: Recognition success dependent on eyes location

subject	23	30	40
01	9747960,5	9954291	28145966
02	12827114,5	10275995	19725206
03	9868985	8900596	27599284
04	7316433	9020885	12682487
05	17986554	19615874	51800432
06	8496437,25	7562373,5	32484104
07	6054513,5	10352797	124337248
08	20266769	28852322	51190396
09	6480843	22042972	57966384
10	9281984	7711191	10051292
mean	10832759,38	13428929,65	41598279,9
mistakes	19	8	110

Figure 3.1: Normalized face using the most suitable settings. From Cal-tech face database.

Table 3.11: Recognition success dependent on eigenfaces number

subject	10	20	30	40	50
01	6914166,5	8803060	9954291	10333164	11223207
02	6146843	8216714,5	10275995	12059686	13489801
03	4201136,5	7677688,5	8900596	10046274	10598203
04	2513733,5	4468037	9020885	11209360	16198316
05	10569339	16690308	19615874	22138186	23770880
06	5091664	5805211,5	7562373,5	9084872	10121169
07	3576268,75	6805508,5	10352797	11876217	15190135
08	18746910	22396306	28852322	30049892	31499254
09	16271612	20306566	22042972	24077264	24756390
10	3969087,5	6427252	7711191	8948959	9979890
mean	7800076,08	10759665,2	13428929,65	14982387,4	16682724,5
normalized	28,198474175	28,092258607	20,333046851	20,078036886	21,279343631
mistakes	51	19	8	26	23

3.3 Summary

So, based on previous experiments, the default values are:

Detector settings

- *Cascade*: haarcascade_frontalface_alt.xml
- *Minimum of neighbors*: 2
- *Scale factor*: 1.1
- *Rotation steps*: 1
- *Maximum image size*: 1600
- *Minimum object width scale*: 0.03
- *Minimum object height scale*: 0.03

Normalizer settings

- *Face width*: 120
- *Face height*: 120
- *Eyes position - X*: 30
- *Eyes position - Y*: 30
- *Half eyes distance*: 0.95
- *Eyes distance from mouth*: 1.4
- *Retinex-surround constant*: 130
- *Blur-surround constant*: 2

First eyes

- *Cascade*: haarcascade_eye.xml
- *Minimum of neighbors*: 20
- *Scale factor*: 1.1
- *Minimum object width scale*: 0.16
- *Minimum object height scale*: 0.16

Second eyes

- *Cascade*: haarcascade_mcs_lefteye.xml
- *Minimum of neighbors*: 20
- *Scale factor*: 1.1
- *Minimum object width scale*: 0.16
- *Minimum object height scale*: 0.16

Third eyes

- *Cascade*: haarcascade_mcs_righteye.xml
- *Minimum of neighbors*: 2
- *Scale factor*: 1.1
- *Minimum object width scale*: 0.16
- *Minimum object height scale*: 0.16

Mouth

- *Cascade*: haarcascade_mcs_mouth.xml
- *Minimum of neighbors*: 40
- *Scale factor*: 1.1
- *Minimum object width scale*: 0.16
- *Minimum object height scale*: 0.16

Nose

- *Cascade*: haarcascade_mcs_nose.xml
- *Minimum of neighbors*: 30
- *Scale factor*: 1.1
- *Minimum object width scale*: 0.16
- *Minimum object height scale*: 0.16

Recognizer settings

- *Eigenface number*: 30
- *Recognize distance*: 22 000 000
- *Distance Metric*: eucleidean

Conclusion

We have implemented a facial recognition system that gives quite good results when used to recognize people present in the initial training set. It has problems when people are not directly facing the camera as well as the results are not accurate when recognizing people which were not included in the training set.

We provided two GUIs, a desktop and a web one to demonstrate some face recognition capabilities.

Here we described face recognition based on eigenfaces, face-feature based object detection and our process of face normalization. Then we provided the results from experiments realized in order to select the best default values.

3.4 Further work

Of course there are things that can be improved. From simple things like face mirroring to obtain twice more images for initial dataset, we can get to complicated problems. It would improve detection if the shadows were somehow removed from the input image.

Problem of our system is that it works properly only on frontal images. 3D face reconstruction would probably solve this problem, since techniques for creating 3D face model from 2D image exist.

Bibliography

[1] Viola, Paul and Jones, Michael J. *Robust Real-Time Face Detection.* International Journal of Computer Vision 57(2), 137-154, 2004

[2] Jobson, Daniel J, Rahman, Zia-ur and Woodell, Glenn A. *A Multiscale Retinex for Bridging the Gap Between Color Images and the Human Observation of Scenes.* IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 6, NO. 7, JULY 1997

[3] *The OpenCV Reference Manual, Release 2.3.* August 17, 2011

[4] *Histogram equalization: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Histogram_equalization&oldid=479270447 [2012-05-03]

[5] *Gaussian Blur: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Gaussian_blur&oldid=484172042 [2012-05-03]

[6] Turk, Matthew and Pentland, Alex *Eigenfaces for recognition.* Journal of Cognitive Neuroscience Volume 3, Number 1

[7] Hewitt, Robin *Seeing With OpenCV: Face Recgnition With Eigenface.* SERVO, APRIL 2007 URL: http://servo.texterity.com/servo/200704?pg=36&search_term=seeing%20with%20opencv&doc_id=-1#pg36

[8] *Principal Component Analysis: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Principal_component_analysis&oldid=489905962 [2012-05-09]

[9] *Covariance: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Covariance&oldid=484980690 [2012-05-09]

[10] *Covariance Matrix: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Covariance_matrix&oldid=490707042 [2012-05-09]

[11] *Google driverless car: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Google_driverless_car&oldid=493925476 [2012-05-24]

[12] *I spy, with my big eye.* The Economist, April 28th 2012

[13] *Facial recognition system: Wikipedia, The Free Encyclopedia.* URL: http://en.wikipedia.org/w/index.php?title=Facial_recognition_system&oldid=492619178 [2012-05-24]

List of Tables

List of Abbreviations

PCA = Principal Component Analysis
DLL = Dynamic-link library
TBB = Threading Building Blocks
GUI = Graphical User Interface
DB = Database
SQL = Structured Query Language
struct = short for *structure*
AJAX = Asynchronous JavaScript and XML
URL = Uniform Resource Locator
RDF = Resource Description Framework
RSS = RDF Site Summary or Really Simple Syndication

Attachment : CD

- Text of the thesis is located in "/thesis.pdf"
- Source codes (whole Visual Studio solution) are in the folder "/Project"
- Desktop application is in folders "/x86" and "/x64", so the user can choose architecture. Executable file is "FaceRecoGUI.exe"
- Sample image dataset (can be used for training) is located in the folder "/CalTech-faces"